Programmation C

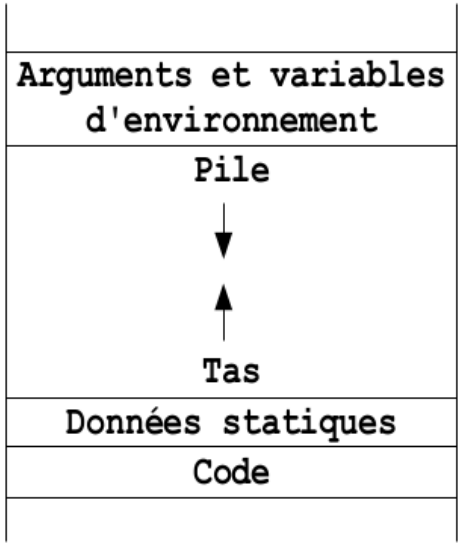
Types, variables et constantes

ESIPE Info 1ère année

Université de Marne-la-Vallée

La mémoire

- Tout programme manipule de la mémoire.
- L'atome est l'**octet** = unité indivisible de 8 bits (et chaque bit vaut 0 ou 1).
- L'art de coder en C : savoir modéliser/diviser/utiliser proprement et efficacement la mémoire pour encoder l'information nécessaire aux programmes. Cette liberté dans la modélisation est une force du langage (exécutable efficace) mais une complexité pour le programmeur (comprendre et savoir représenter).
- Mémoire utilisée par un programme :



Les types en langage C

- type = taille de zone mémoire + interprétation (signé ou non / nombre entier ou flottant)
- dépendant de la machine (et du système installé dessus...)
  - taille des pointeurs (c'est-à-dire des adresses) :
  - 4 Octets sur une machine 32 bits
  - 8 Octets sur une machine 64 bits
- seule contrainte du langage C pour les tailles :

$$\texttt{sizeof(char)} \leqslant \texttt{sizeof(short)} \leqslant \texttt{sizeof(int)} \leqslant \texttt{sizeof(long)} = \texttt{sizeof(size_t)}$$

Les types entiers de base

type	taille	signé	valeurs
signed char	1	oui	[-127, 127]
unsigned char	1	non	[0, 255]
char	1	Ça dépend du compilateur !	
(signed) int	4	oui	[-2 147 483 647, 2 147 483 647]
unsigned int	4	non	[0, 4 294 967 295]

- La norme C contraint les intervalles des types signés entre $-(2^N - 1)$ et $2^N - 1$.
- Un compilateur **peut** autoriser la valeur -2^N (par exemple, un char valant -128).
  $\Rightarrow$ Portabilité compromise *(Je vous jure monsieur, ça marchait chez moi...)*

Débordement des types entiers

- Aucune vérification ! (force : plus rapide, faiblesse : n'attrape pas les erreurs)

```
1 unsigned char c = 255;
2 c = c + 1;
```

$\Rightarrow$ c vaut finalement 0.

- Cela peut provoquer des boucles infinies (tellement plus embêtantes que les Segfaults)

```
1 void count_down(int n){
2   unsigned int i;
3   for (i=n ; i>=0 ; i--){
4     printf("%d\n", i);
5   }
6 }
```

Représentations des entiers

- décimale : 1234
- hexadécimale : `0x4D2`
- octale : 02322
- Pas de représentation binaire ! (pénible, faut le faire à la main!)

Opérateurs sur les entiers

- addition : `9+4` (13)
- soustraction : `9-4` (5)
- multiplication : `9*4` (36)
- quotient de la division entière : `9/4` (2)
- reste de la division entière : `9%4` (1)

L'expression `i++`

- ne peut pas s'appliquer sur une constante
- utilise la valeur courante de i, puis l'incrémente de 1

```
1 int i = 2;
2 int a = i++;
```

a vaut 2 et i vaut 3

- à utiliser avec précaution !!!

```
1 int a = (i++*2+1)*(i++);
```

valeur de a ???

```
1     if ((test_horrible()) && (a++))
```

Quand est-ce que la valeur de a est modifiée ?

L'expression `++i`

- `++i` : idem mais on incrémente d'abord

```
1 int i = 2;
2 int a = ++i;
```

a et i valent 3

- `i--` et `--i` : même chose avec la soustraction

Les types réels

- Calcul en virgule flottante
  $\Rightarrow$ approximations (ne pas utiliser pour les calculs exacts, ex: finances (ne le dites pas qu'ils le font...))
- `float` : réel simple précision
- `double` : réel double précision

Représentation des réels

- 12.34 ou 1234. ou .1234
- notation *mantisse* $\times 10^{exposant}$
  1723.68 = 1.72368e3 = 17.2368E2
  0.015 = 1.5e-2
- par défaut, les constantes sont de type double
  245.45f = 245.45F $\Rightarrow$ `float`
  245.45 = 245.45l = 245.45L $\Rightarrow$ `double`
- environ 7/15 chiffres significatifs pour les float / double

Problèmes de précision

- se méfier beaucoup des `float`
- se méfier quand même des `double`

```
1  int main(int argc, char* argv[]){
2    float f = 0.f;
3    double d = 0;
4    int i;
5    for (i=0 ; i<1000 ; i++)
6      f = f + 0.1f;
7    for (i=0 ; i<10000000 ; i++)
8      d = d + 0.1;
9    printf("f = %f\nd = %f\n", f,d);
10   return 0;
11 }
```

affiche au final

```
f = 99.999046
d = 1000000.000001
```

C’est pas rien deux fois rien !

L’addition des flottants n’est pas associative. A partir de là, toute l’arithmétique est à surveiller de près.

```
1  int main(int argc, char* argv[]){
2    double d = 67482746.433251;  /* un grand */
3    double rien = 0.000000249;   /* un petit */
4    printf("deux fois rien : %f\n", (d + rien) + rien);
5    printf("deux fois rien : %f\n", d + (rien + rien));
6    return 0;
7  }
```

affiche au final (oui je l’ai fait exprès, mais ça ne prévient pas quand ça bogue un programme...)

```
deux fois rien : 67482746.433252
deux fois rien : 67482746.433251
```

Opérations sur les réels

- `+` `-` `*` comme pour les entiers
- `/` comme une division réelle
- conversions automatiques :

  `2` `+` / `-` / `*` `3.5` ⇒ `2.0` `+` / `-` / `*` `3.5`

  `6 / 1.5` ⇒ `6.0 / 1.5`

  `8.4 / 2` ⇒ `8.4 / 2.0`

  `double a=2;` ⇒ `double a=2.0;`

  `int a=2.45;` ⇒ arrondi à `int a=2;`
- pour obtenir un résultat réel lors de la division de deux variables de type entiers, il faut une conversion explicite avant l'application de la division

Le type caractère

- on interprète les `char` comme des codes de caractères
  - pas de problème entre 0 et 127
  - au delà, dépend de l'encodage du système (à éviter)
- on désigne le code du caractère `x` par `'x'`
- on peut utiliser les opérateurs entiers
  - exemple: `'a'+1` vaut `'b'` (car les codes sont bien rangés)
- ne pas confondre caractère et valeur d'un chiffre (le char `'9'` n'est pas indexé par l'entier 9 dans la table ascii)
- Et comme d'habitude, le juge de paix `man ascii` !

Caractères spéciaux

- `'\n'` : saut de ligne
- `'\r'` : retour au début de ligne
- `'\t'` : tabulation (largeur variable)
- + `'\\'` et `'\ "'` pour les caractères `\` et `"`

Les chaînes de caractères

- Ce n'est pas un type mais une convention de stockage (mémoire contiguë contenant des `char`) : tableau de caractères dont le dernier '\0'

- délimitées par des guillemets
```
char* msg="Welcome";
```

- variantes :
```
char* msg[]="Welcome";
char*
msg[]={'W','e','l','c','o','m','e','\0'};
```

Les chaînes de caractères

- chaîne vide "" ⇒ tableau dont la première case contient '\0'
- longueur = nombre de caractères avant '\0'
  "abcd" a une longueur de 4
- concaténation automatique des chaînes constantes

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]){
4    printf("aaaa\n"
5           "bbbb\n"
6           "cccc\n");
7
8    return 0;
9 }
```

Les chaînes de caractères

- accès au $n^{\text{ième}}$ caractère :

  `char* msg="Welcome";`
  `char c=msg[2];`
  $\Rightarrow$ `c` vaut `'l'` car on commence à l'indice 0

- attention aux débordements

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]){
4   char* s = "Hello";
5   printf("%c\n", s[6354]);
6   return 0;
7 }
```

donne une jolie erreur de segmentation...

Les chaînes de caractères

- On ne peut ni modifier les chaînes statistiques, ni les arguments de `main` !

```
1 int main(int argc, char* argv[]){
2   char* s="abcd";
3   s[2] = 'u';
4   printf("%s\n", s);
5   return 0;
6 }
```

BOOOUUUMMMMM!!!!!!!!!!

```
1 int main(int argc, char* argv[]){
2   argv[0][0] = '*';
3   printf("%s\n", argv[0]);
4   return 0;
5 }
```

PAS BOUM! (mais pas bien...)

Les variables

- identificateurs : `[_a-zA-Z][_a-zA-Z0-9]*`
  (même règle que pour les noms de fonctions!)
  `Hello_89`, `__foo`, `b_l_a`, ~~`3tree`~~

- éviter les noms qui ressemblent à des mots-clés: `new`, `class`...

- appeler un chat un chat
  un compteur peut facilement se déclarer
  `int compteur=0;`
  (code parlant = documentation allégée !!!)

  Donner des noms explicites aux variables et aux fonctions !!!

Les variables

- doivent être déclarées avant d'être utilisées
- déclaration : `int a;`
- avec initialisation : `int a=3;`
- déclarations multiples : `float f1=1.3, f2;`
- attention :

```
1 #include <stdio.h>
2
3 int main(int argc, char* argv[]){
4    char* a="Hello", b="you";
5    printf("%s␣%s", a, b);
6    return 0;
7 }
```

donne

```
initialization makes integer from pointer without a cast
```

Portée d'une variable

- visible dans le bloc : A NE PAS FAIRE !!!

```
1 int main(int argc, char* argv[]){
2   int i;
3   for (i=0, i<10, i++){
4     int a = i;  /* Horrible !!! */
5   }
6   printf("a=%d\n", a);
7   return 0;
8 }
```

- dans une fonction : variable locale
- dans un fichier : variable globale

Variable globales

- utilisable dans tout le fichier
- plutôt à éviter :
  - nuit à la modularité
  - pas thread-safe
- Dans quelques cas, on peut s'autoriser une variable `static` sur un fichier

```
1  static int nb_echange = 0;
2
3  void tri_selection(...){
4     ...
5     nb_echange++;
6     ...
7  }
8
9  void tri_insertion(...){
10    ...
11    nb_echange++;
12    ...
13 }
```

Initialisation des variables

il faut toujours initialiser les variables de façon que l'exécution ne dépende pas des valeurs par défaut.

```
1  int sum(int n){
2    int i, x=0;
3    while (i<n){
4      x = x+1;
5      i++;
6    }
7    return x;
8  }
```

Le résultat dépend de la valeur par défaut de `i`.

```
1  int sum(int n){
2    int i, x=0;
3    for (i=0 ; i<n ; i++){
4      x = x+i;
5    }
6    return x;
7  }
```

`i` est ici initialisé avant utilisation par le `for`.

Affectation

- `=` : est un opérateur !
  `int a=3;` ⇒ déclare un entier `a` qui aura la valeur 3.

- l'opérateur `=` affecte une valeur à une variable mais a aussi une valeur de retour (qui est la valeur affectée)
  `int a=3;`
  `int b=(a=a+1)+3;` ⇒ `a` vaut 4 et `b` vaut 7

- pratique pour tester les retours de fonction

```
1  if ((p = (int*)malloc(5*sizeof(int))) == NULL){
2    fprintf(stderr, "Erreur d'allocation\n");
3  }
```

  test propre de retour de `malloc`

Effets de bord

- modification d'une valeur pendant l'évaluation d'une fonction
- dangereux
- à utiliser que dans des cas simples

```
1  if ((b == 3) && (a++ == 0))
```

Horrible!!! Quand est-ce que a est incrémenté ?

Conversion

- `unsigned char c = 713;`

  $\Rightarrow$ (un des warning de `-Wall` s'active) attention : grand entier implicitement tronqué pour un type non signé [-Woverflow]

  $\Rightarrow$ `c = 201`

- ce qui se passe en binaire :

713	=	1	0	1	1	0	0	1	0	0	1
201	=			1	1	0	0	1	0	0	1
					⟵		8 bits		⟶		

- on évite les warning avec des cast :

  `variable = (type) expression`

- attention, il faut être sur de ce que l'on fait avec les cast

Promotion entière

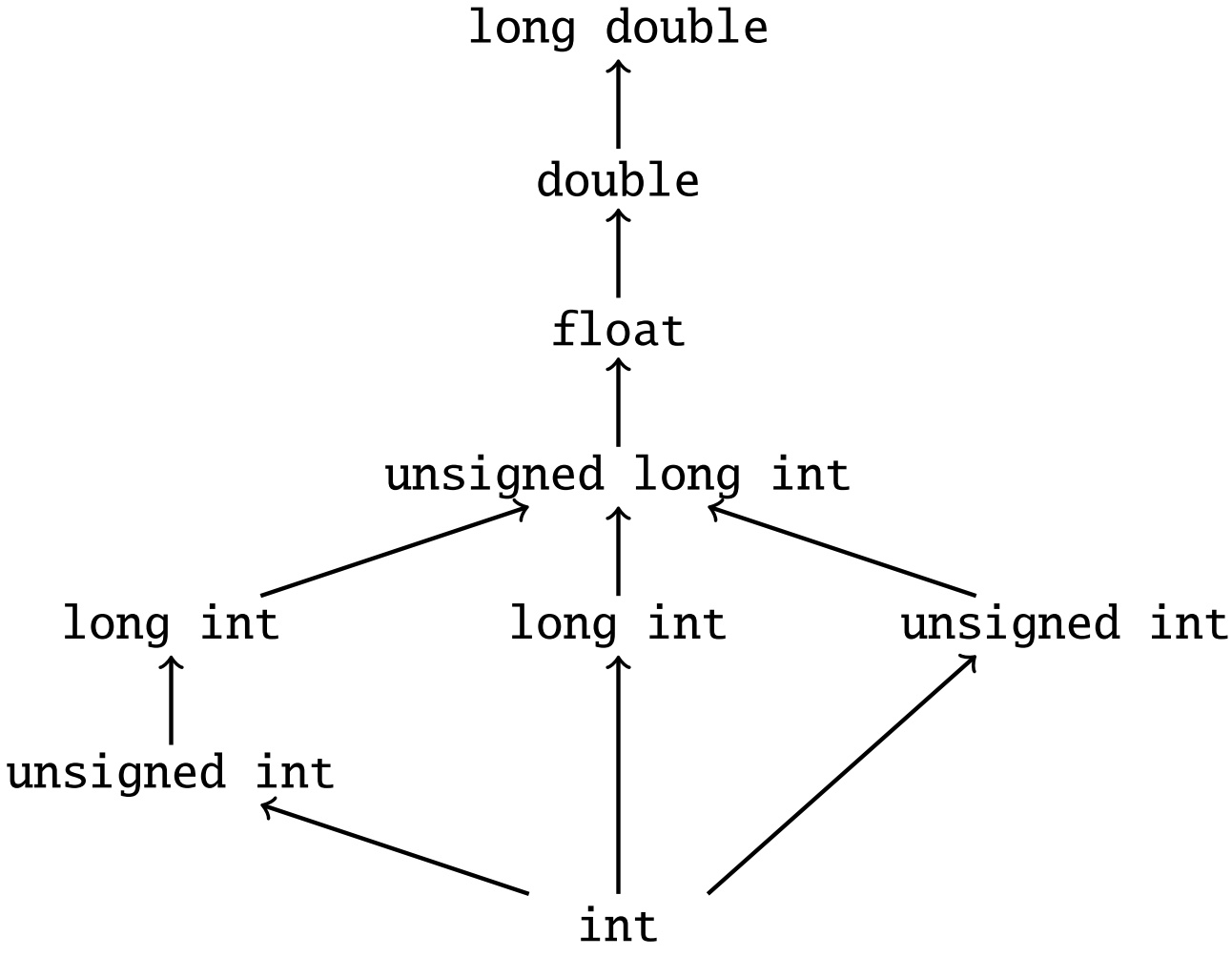
Lorsque l'on fait une opération binaire avec deux types entiers, le résultat se place dans un type pouvant contenir les deux données.

(`signed` ou `unsigned`)(`char` ou `short int`) $\longrightarrow$ (`int` ou `unsigned int`)

Règle : si l'entier de départ est représentable dans le type de destination, sa valeur reste inchangée sinon la conversion est machine dépendante.

Encore une fois, attention aux cast et aux plages de valeurs possibles pour les variables castées !

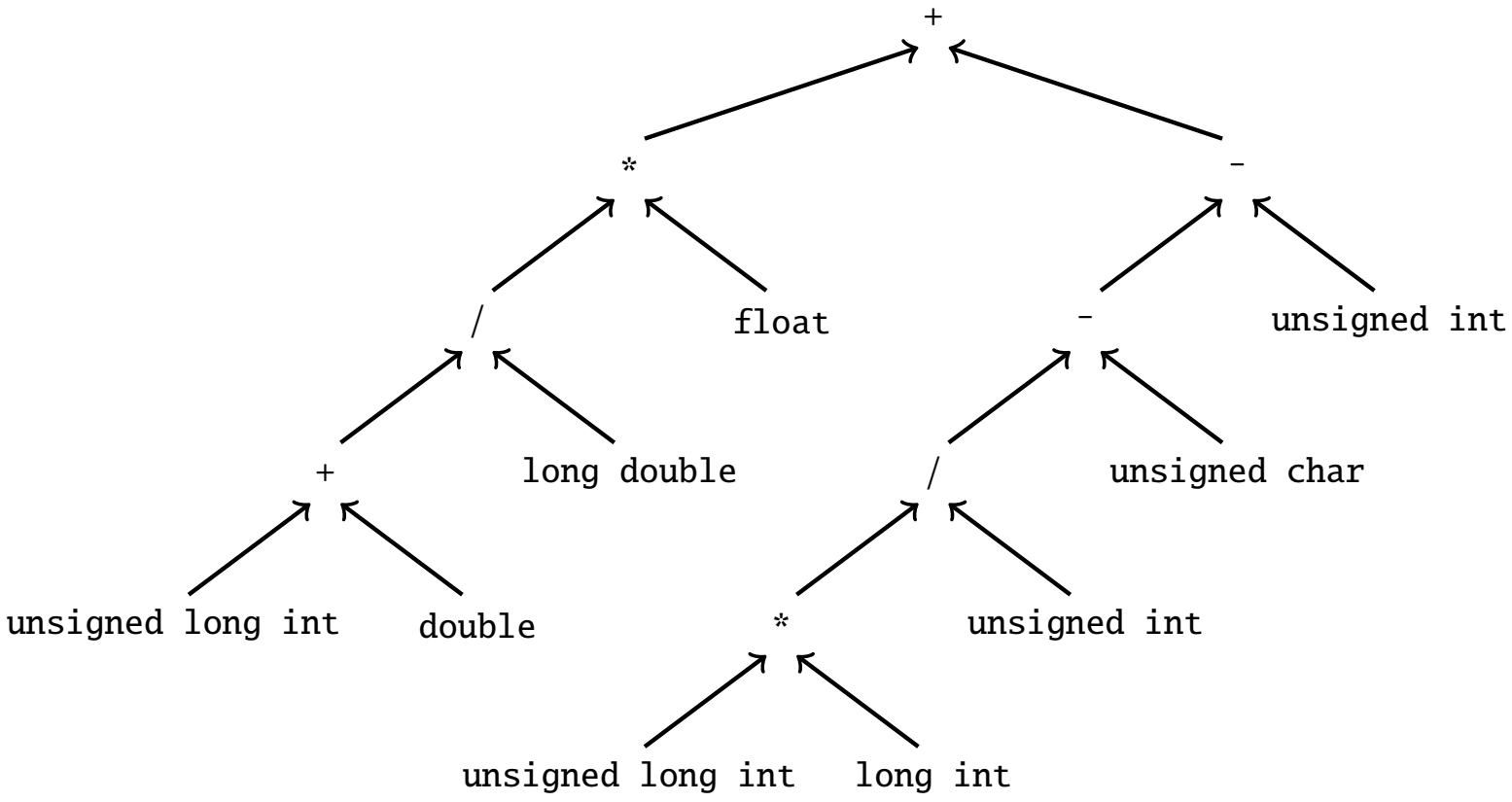
Conversions arithmétiques



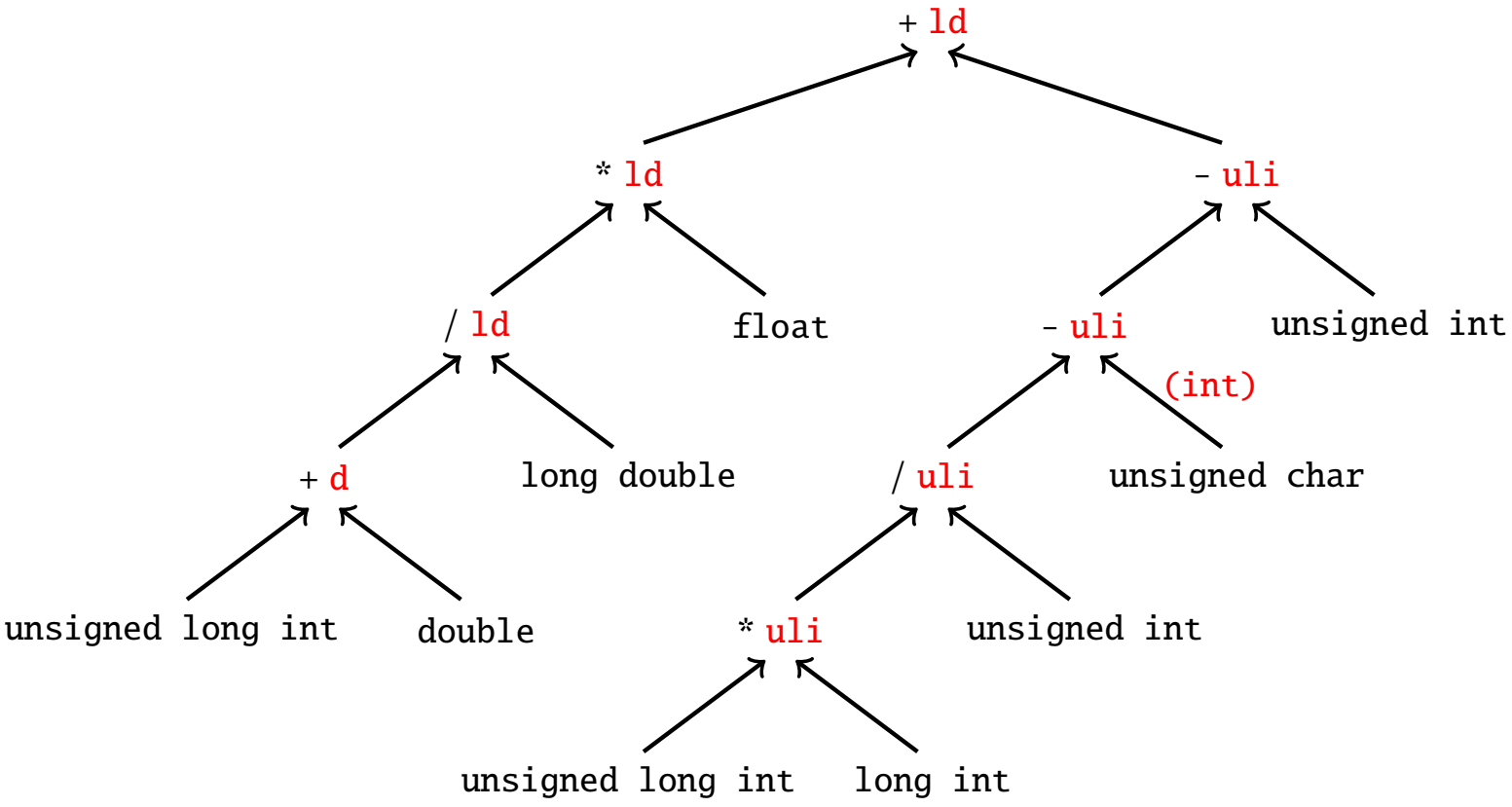
L'utilisation de `unsigned` peut rendre les conversions machines dépendantes lorsque `long int` contient `unsigned int`.

Exemple

Donner le type de:



Correction


+ ld
* ld
- uli
/ ld
float
- uli
unsigned int
(int)
+ d
long double
/ uli
unsigned char
unsigned long int
double
* uli
unsigned int
unsigned long int
long int

Comportements machines dépendants

- arrondi de / : machine dépendant pour les opérandes négatives
- signe de % : machine dépendant pour les opérandes négatives
- division par 0 : comportement machine dépendant
- les dépassements de capacité

Du bon usage des entiers : conclusions

- On préférera utiliser le type `int` pour les entiers.
- Les types `char` et `short` sont utilisés uniquement pour économiser la mémoire, seulement quand c'est absolument nécessaire.
- Pour faire des opérations de manipulation de bits (on reverra cela plus tard), on utilisera de préférence `unsigned`.
- Pour stocker des valeurs positives ou nulles, on pourra utiliser `unsigned` si les opérations arithmétiques utilisées derrière restent simples.

Les constantes

- définies avec la commande :

```
#define IDENTIFICATEUR valeur
```

- exemples :

```
#define PI 3.14
#define SIZE_MAX 1024
#define YES 'y'
#define PROMPT "$>"
```

Les constantes

- remplacement brut de chaînes par le préprocesseur sauf à l'intérieur strict des identificateurs et dans les chaînes

- risques d'erreur

```
1  #define A 123
2
3  int main(int argc, char* argv[]){
4     int A = 12, Ab = 12;
5     printf("A = %d\n", A);
6     return 0;
7  }
```

donne après action du préprocesseur

```
1  int main(int argc, char* argv[]){
2     int 123 = 12, Ab = 12;
3     printf("A = %d\n", 123);
4     return 0;
5  }
```

Les constantes

- toute constante non triviale doit être définie
- leur usage évite les modifications multiples
- donner des commentaires sur les constantes
- bien faire attention aux collisions (d'où l'utilisation de majuscules) avec les autres identificateurs